Lazy Threads: Implementing a Fast Parallel Call

Seth Copen Goldstein
Computer Science Division, University of California–Berkeley

Klaus Erik Schauser
Department of Computer Science, University of California–Santa Barbara

David E. Culler
Computer Science Division, University of California–Berkeley

Lazy Threads: Implementing a Fast Parallel Call

Seth Copen Goldstein
Computer Science Division, University of California–Berkeley
Berkeley, CA 94720
510-642-8299
sethg@cs.berkeley.edu

Abstract

In this paper we describe lazy threads, a new approach for implementing multi-threaded execution models on conventional machines. We show how they can implement a parallel call at nearly the efficiency of a sequential call. The central idea is to specialize the representation of a parallel call so that it can execute as a parallel-ready sequential call. This allows excess parallelism to degrade into sequential calls with the attendant efficient stack management and direct transfer of control and data, yet a call that truly needs to execute in parallel, gets its own thread of control. The efficiency of lazy threads is achieved through a careful attention to storage management and a code generation strategy that allows us to represent potential parallel work with no overhead.

1 Introduction

Many modern parallel languages provide methods for dynamically creating multiple independent threads of control, such as forks, parallel calls, futures, object methods, and non-strict evaluation of argument expressions. These threads describe the *logical parallelism* in the program. The language implementation maps the dynamic collection of threads onto a set of physical processors executing the program, either by providing its own language-specific scheduling mechanisms or by using a general threads package. These languages stand in contrast to languages with a single logical thread of control, such as High Performance Fortran [22], or a fixed set of threads, such as Split-C [8] or MPI [12]. There are many reasons to have the logical parallelism of the program exceed the physical parallelism of the machine, including ease of expression and better resource utilization in the presence of synchronization delays, load imbalance, and long communication latency [26, 39]. Moreover, the semantics of the language or the synchronization primitives may allow dependencies to be expressed in such a way that progress can be made only by interleaving multiple threads, effectively running them in parallel even on a single processor [28].

Regardless of how the dynamic logical parallelism is expressed at the language level, the underlying execution model has two key features. First, the control model supports the dynamic creation of multiple threads with independent lifetimes. Second, each thread may require an unbounded stack. Thus the storage model is a tree of stacks, called a *cactus stack*. Unfortunately, a parallel call or thread fork is fundamentally more expensive than a sequential call because of the thread and storage management, data transfer, scheduling, and synchronization involved. Previous work has sought to reduce this cost by using a combination of compiler techniques and clever run-time representations [9, 21, 26, 29, 30, 33, 35, 37, 39], and by supporting fine-grained parallel execution directly in hardware [3, 19, 31]. In many cases, the cost of the fork is reduced by severely restricting what can be done in a thread. These approaches, among others, have been used in implementing parallel programming languages such as ABCL [40], CC++ [6], Charm [20], Cid [29], Cilk [4], Concert [21], Id90 [9, 30], Mul-T [23], and Olden [5]. Still, a fork remains substantially more expensive than a simple sequential call.

Our goal is to support an unrestricted parallel thread model and yet bring the cost of thread creation, termination, and switching down to essentially the cost of a sequential call. We observe that fine-grained parallel languages promote the use of small threads that are often short lived—sometimes on the order of a single function call. Fortunately, logically parallel calls can often be run as sequential calls. For example, once all the processors are busy, there may be no need to spawn additional work, and in the vast majority of cases the logic of the program permits the child to run to completion while the parent is suspended. Thus a parallel call should be viewed as a *potentially parallel call*. It is the point in the computation where an independent thread may be, but is not necessarily required to be created. This property has previously been exploited through load-based inlining and Lazy Task Creation (LTC), which attempt to execute parallel calls sequentially when parallelism is not required [26]. In this paper, we go much further, performing a potentially parallel call almost exactly like a stack-based sequential call. Our code generation strategy avoids creating task descriptors, initializing synchronization variables, or even explicitly enqueuing tasks. Through careful attention to the program representation, we pay almost nothing for the ability to elevate a sequential call into a full thread on demand. We call this overall approach *Lazy Threads*.

The fundamental idea behind lazy threads is to implement the potentially parallel call as a *parallel-ready sequential call*. The call allocates the child frame on the stack of the parent, like a sequential call. Control is transferred directly to the child (suspending the parent), arguments are transferred in registers, and, if the child returns without suspending, results are transferred in registers when control is returned to the parent; just like a sequential call. Moreover, even if the child suspends, the parent is resumed and can continue to execute without needing to copy the stack. Instead, the suspending child assumes control of the parent's stack and further children called by the parent are executed on stacks of their own. In other words, the suspending child steals the parent's thread. Additionally, if work is needed by another processor, the parent can be resumed to spawn children on that processor. The advantage of lazy threads is that when they run sequentially (i.e., when they run to completion without suspending) they have the same efficiency as a sequential call. Yet, the cost of elevating a lazy thread into an independent thread of control is close to that of executing it in parallel outright. This contrasts with pervious approaches to lazy parallelism based on continuation stealing in which the parent continues in its own thread forcing stack copies and migration.

In the rest of the paper we present the run-time data structures and compilation techniques necessary to reduce the overhead for thread creation and manipulation sufficiently to allow unfettered use of multiple threads, as in fine-grained parallel languages.

In Section 2 we describe the three main aspects of a parallel call: how the child is invoked, how it returns, and how it is scheduled. Invocation has three independent actions: transferring arguments,

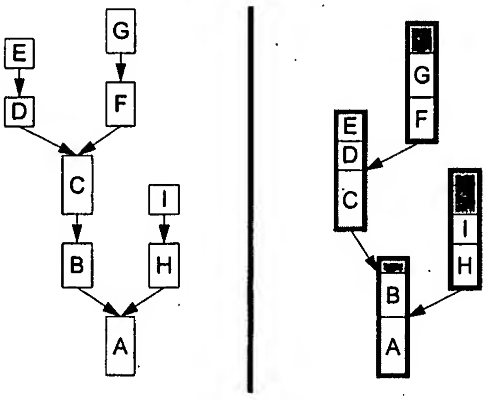

Figure 1: How individual activation frames of a cactus stack are mapped onto stacklets. Only parallel calls or stacklet overflows require allocation of a new stacklet. (The arrows point back to the parent. In the above example, A calls B and H in parallel.)

initiating the child, and suspending the parent. These are combined in a sequential call. There are also three independent actions that occur when a child returns to its parent: storing the results, updating the parent's synchronization state, and finally determining the continuation in the parent. Most of the complexity in creating a low-overhead parallel-ready sequential call occurs in trying to combine these three actions into one, as in the sequential case. We develop a *control hierarchy*, a hierarchy of representations for the call with varying cost and generality, allowing the compiler to select the most appropriate and least expensive representation. We are able to achieve most of the sequential efficiency in both the call and return mechanisms because we rely on the invariants created by following the sequential scheduling order. We finish Section 2 with examples of how the context of the call site affects which representation is chosen for the parallel call.

Given this general understanding of the parallel-ready sequential call, we proceed to explain how the underlying storage model and control model are implemented. First, in Section 3, we address storage allocation. Since threads can fork other threads and each requires a stack, a tree of stacks or cactus stack is required. We realize this cactus stack using *stacklets* (see Figure 1), which allow activation frames to be allocated by adjusting the stack pointer. Allocation of a new stacklet occurs when a new thread is created or when a stacklet overflows. Stacklets alone provides a naive parallel language implementation with conventional local and remote eager forks, i.e. where every fork realizes a new thread.

In Section 4, we develop the control mechanisms needed for the parallel-ready sequential call . We begin by introducing a lazy thread fork, which addresses control and data transfer when a thread is forked on the local processor. Next, we reduce the overhead in returning to the parent by combining two of the three independent return actions (storing the results and finding the parent's continuation address) using *parent controlled return continuations*. Finally, we combine all three return actions into one using *synchronizers*.

In Section 5 we develop the remaining portions of the control hierarchy. Using the invariants inherent in the parallel-ready sequential call we show how these representations can be used to find additional parallel work on demand, in response to a work-stealing request from another processor. We then examine the costs of the different call representations.

In Section 6, we present empirical data to show that these concepts can be combined to efficiently implement dynamic logical parallelism. Our experimental study focuses on two prototype implementations on the CM-5: a direct implementation in C and a compiler for the fine-grained parallel language Id90. The C implementation shows that these primitives introduce little or no overhead over sequential programs. The Id90 implementation shows that for complete programs we achieve a substantial improvement over previous work. Our techniques can be applied to other programming languages [6, 40], thread packages [11], and multithreaded execution models. In Section 7 we discuss related work.

Our work, Lazy Threads, relies extensively on compiler optimizations and cannot simply be implemented with function calls to a user-level threads library without substantial loss of efficiency. Underlying our optimizations is the observation that in modern microprocessors, a substantial cost is paid for memory references and branches, whereas register operations are essentially free. Since a stacklet is managed like a sequential stack, arguments and results can be passed in registers, even in the potentially parallel case. By manipulating the existing indirect return jump, conditional tests for synchronization and special cases can be avoided.

2 The Potentially Parallel Call

This section describes a spectrum of representations for potentially parallel calls. In short, the compiler chooses for every fork the least expensive implementation that meets the semantic requirements of that particular call. In this work a *thread* is a locus of control on a processor (or in an operating system process) which can perform calls to arbitrary nesting depth, suspend at any point, and fork additional threads. Threads are scheduled independently and are non-preemptive.[1] We associate with each thread its own logically unbounded stack. Each thread is initiated by a fork, i.e., a potentially parallel call, and synchronizes with a join in its parent.

Before considering the parallel call, observe that the efficiency of a sequential call derives from two cooperating factors. First, the parent is suspended upon call and all of its descendants have completed on return. Second, data and control are transferred together on call and return. The first condition implies that storage allocation for the activation frames involves only adjusting the stack pointer. The second condition means that arguments and return values can be passed in registers and that no explicit synchronization is required.

In an idealized parallel execution model, every fork is executed in its own processor, in other words, each fork is executed eagerly. Where this logical parallelism is not required, it is advantageous to execute collections of threads sequentially on the local processor. However, in general one cannot decide at compile time which threads need to run in parallel. We must therefore represent the fork such that it can run in parallel if necessary. Our goal is to minimize the overhead of this representation without reducing the amount of parallelism available to the application.

A previous approach, load-based inlining, provides both an eager parallel call and a sequential call for each fork. When a potentially parallel call is encountered the load characteristics of the parallel machine are used to decide whether to execute it sequentially (inline it) or execute it in parallel [23]. Unfortunately, these decisions introduce overhead and are irrevocable, which can lead to serious load imbalances or deadlock.

Another approach, taken by Lazy Task Creation, is to perform the fork like a sequential call, but record the point of execution after the fork (i.e. the continuation) in the parent. If the child suspends or if more parallelism is required to keep processors busy, the continuation is stolen and a copy is made of the parent's stack and the parent's thread is resumed or migrated to the copy. This requires bookkeeping at every fork and the ability to migrate stack frames to new threads, potentially across processors. In order to perform such migration one must forbid pointers into stack frames. This is particularly onerous on distributed memory machines since it restricts such latency hiding operations as pipelining remote memory operations. Additionally, in distributed memory machines local heap pointers are not valid when moved to other processors. In many cases migrating an already created frame, can lead to poor data locality. For example, if the parent is migrated after it has started execution any data structures created by it or its children will then cause remote accesses.

Observe that if after a child suspends and the parent is resumed forward progress is made by spawning off additional threads then migration is not required. To prepare the parent to continue without requiring migration we can create representations for the remaining forks in the parent which can later be instantiated either sequentially or in parallel. When a child suspends or a remote work request arrives we can instantiate the representation as a full fledged thread. These representations of the remaining forks are *nascent threads*, or, if run sequentially, nascent activation frames. Moreover, we do not really have to build a descriptor for a nascent thread, because all the information that would be put into the descriptor is already in the stack frame of the parent. All we really need is a means of locating and invoking the code in the parent that performs the fork. Because the cost of creating these representations is nominal (or even free), they allow us to produce parallelism on demand without the overhead normally associated with parallel calls.

2.1 Requirements of Return

A potentially parallel call must provide the semantics of a parallel call even if it is run like a sequential call in the same thread as its parent. We identify three independent actions that occur when a parallel call returns. First, the child must be able to return results to the parent so that the parent can use them in the future. We call the code that stores the results an *inlet*. Second, the parent must update the synchronization status when the call is issued and then when it returns. Finally, a continuation address must be maintained in the parent to indicate what actions need to be performed after the call has completed.

For a sequential call, these three actions are not independent and all occur at the return address of the function. This is because sequential calls only return to the parent when they have completed. The return address points to the inlet. No synchronization state is needed since the parent only resumes when the child has completed. Following the inlet code is the code for the rest of the function, in other words, the return address is also the continuation address.

To reduce the overhead of a fork to that of a sequential call when the logical parallelism represented by the fork is not needed, we must combine the above three independent actions such that we can still elevate a sequentially invoked fork into its own thread. Suppose there are n consecutive forks followed by a join. The main difficulty is that at the point of invoking any but the last fork the continuation address can not point to the join, but must point to the fork that follows the one being invoked. Indeed, if the fork executes sequentially the return address can work as does the sequential call. However, if the child suspends, or some other fork is started while the child is still running (e.g., some other processor steals work), the child must

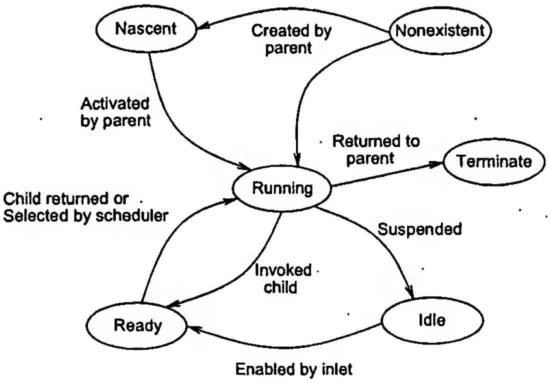

Figure 2: The legal state transitions for a task.

now have an independent inlet address and the continuation address in the parent has to be updated.

The above argument shows that we cannot use a sequential call to represent a potentially parallel call. One can think of many ways to begin a fork as a sequential call and then if necessary convert it into a parallel call. All the methods will have to create a new thread (for either the suspended call or its parent) and turn the original return address into both an inlet address and a continuation address, update the synchronization state of the parent, and finally modify the parent to account for the fact that its child is now a parallel task. This general method can be used both for children that suspend and children that are migrated to another processor.

2.2 Scheduling Frames

Before describing the variety of representations for the potentially parallel call we need to define how frames are scheduled. Every potentially parallel call is considered to start a logically independent thread. There is no preemption. Fairness is not an issue because in order for the entire application to finish all the tasks will need to run to completion.

Each task is identified by its activation frame (or frame), analogous to a stack frame in a sequential language. A frame can be in one of four states: running, ready, idle, and nascent. There is exactly one running frame per processor. A ready frame is a one that the scheduler may run when the currently running frame becomes idle. An idle frame is waiting on some event to become ready. A nascent frame is not truly a frame, but some representation of the task that is able to become a frame. The state transitions for a frame are shown in Figure 2.

Frames give up the processor in one of three ways: they call a potentially parallel child, they suspend, or they return to their parent. For the first and last of these we mimic the way in which control is transferred in sequential stack-based languages, because it allows us to use registers for transferring arguments between parent and child, and for transferring results between the child and the parent. Furthermore, if a fork executes sequentially we expect to get the same performance from the fork as we would from a sequential call. Thus, when a frame invokes a child it is logically making the state transition from 'running' to 'ready,' but instead of putting the parent on an explicit ready queue, we use the activation frames in the cactus stack as an implicit ready queue.

When a frame suspends there are four possible choices for the next frame to run: its parent frame, a frame from the ready queue, a nascent frame, or an explicitly selected frame. The choice of which frame to run has a significant impact on the overall performance of the system and the amount of resources consumed by the application. The default action is to follow the sequential scheduling order and run the parent. Thus, the cactus stack is used as an implicit scheduling queue of ready frames. If the child chooses to run a frame other than its parent, it must put its parent on an explicit ready queue.

The transition from 'nascent' to 'running' occurs when a nascent frame is activated, or in other words, when the representation for the nascent frame is instantiated as a real frame. The transition from 'ready' to 'running' occurs when a child returns or suspends and transfers control to its parent, or when the system scheduler picks a frame from the ready queue. The transition from 'running' to 'idle' happens when a frame suspends. Finally, the transition from 'idle' to 'ready' occurs when an inlet runs and enables the frame, enqueueing it onto an explicit ready queue.

We also impose an implied scheduling order to consecutive forks. Forks are spawned off (with whatever representation is chosen for them) in the order they appear lexically. Thus, if they do run sequentially, the first starts, then it returns, followed by the second starting, etc. They follow this order even if a child suspends. Thus, when the i^{th} child suspends (or returns) the $i + 1^{st}$ child will be invoked next. If the i^{th} child chooses not to return to the parent, then it must enqueue the parent on an explicit queue so that the $i + 1^{st}$ (and all the rest) of the children can be called.

At this point we have identified two ready queues: one implicit and one explicit. The implicit queue is embedded in the cactus stack and is composed of the parents that are 'ready' and the nascent frames. The explicit queue contains those frames that suspended and were later explicitly enqueued by inlets. As we shall see shortly, we will need one more explicit queue used for nascent frames, which is called the *seed*

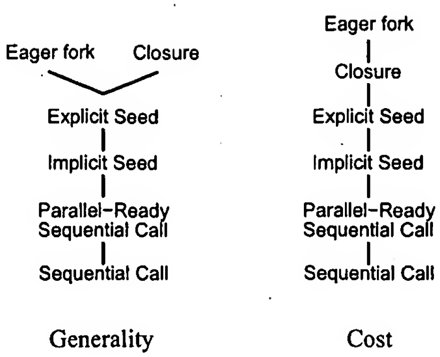

Figure 3: The hierarchies of costs and generality for potentially parallel call implementations. The overhead is based on the assumption that the call could have executed sequentially.

queue.

2.3 The Representations

The hierarchy of representations of potentially parallel calls spans the spectrum from the eager parallel call to parallel-ready sequential call. In between these two extremes we have, full closures, *explicit seeds*, and *implicit seeds*, described below in decreasing cost and generality. (See Figure 3.) At compile time the cheapest representation is chosen that is sufficient to the context of the fork.

A full closure is the most general representation of potentially parallel work. It is used when the locations containing the arguments to the fork may be changed by the parent before the closure is activated. Any data that can be potentially modified is extracted from the frame and stored in the closure along with a continuation into the parent which points to a code fragment that uses the closure to execute the fork. The closure is then enqueued on the seed queue.

When the arguments to the fork cannot change between the creation of the representation and the time it is instantiated, we do not need a full closure. Observe that all the arguments are in the activation frame already, so we can represent the fork without creating a full closure. An *explicit seed* is a specialized closure that contains only an instruction pointer. It is stored in the frame of the function that contains the fork being represented. A pointer to the frame location is put on the seed queue. The instruction pointer points to the code fragment generated by the compiler for the fork. This code fragment is generated by the compiler for the sole purpose of scheduling the fork on either another processor or a separate thread. It also manages the parent's synchronization state. Since the code executes in the context of the parent of the fork we do not need to store any of the arguments explicitly. Of course, this representation can only be used when the parent does not change the arguments between the creation of the explicit seed and the last possible point when it could be executed. In short, the compiler generates both the code for the user program and additional code to handle scheduling.

By using the frame storage model and the scheduling implied by sequential calls we can eliminate the need to enqueue the seed. We call these *implicit seeds*. First, we observe that if a fork is executed as a parallel-ready sequential call we can rely on the fact that it will return to its parent when it has completed. We also guarantee that if the child suspends it will return to the parent. An implicit seed is just the return address of the child.

The implicit seed may only be used for a fork that immediately follows another fork. This constraint arises because we use the return address of the preceding call as the implicit seed. In other words, we extend the concept of the return address to include a pointer to the compiler generated code fragment which can start the second fork.

2.4 Patterns of Parallel Calls

For each call site appearing in the program, the compiler selects the best representation at the call based on the context in which it appears. Here we describe the representations that would be used for four basic patterns of fork usage: n consecutive forks, two consecutive forks, one fork followed by sequential code, and finally a parallel loop; which cover the most important cases occurring in practice.

The canonical pattern for task parallelism is n consecutive forks followed by a join. This pattern can be compiled using a parallel-ready sequential call for the first fork and one seed (either implicit or explicit) for the following $n - 1$ forks. Each time a seed is instantiated one seed is used to represent the remaining forks. We do not need to use a closure for the subsequent forks because the seed will only be executed in the context of the parent; which already has storage for the arguments in the frame. The choice of an implicit or explicit seed affects the ease of performing work stealing. On a single processor system an implicit seed would be sufficient. In the special case when $n = 2$ we may further reduce the cost for explicit synchronization as described in Section 4.3.

The more difficult case is when a single fork is followed by sequential code and then a join. In this case, if we used a parallel-ready sequential call to represent the fork, we would lose the parallelism between the forked child and the parent's sequential code. We lose the parallelism because both the child and the parent are executed on the same processor and don't allow migration. If we allowed migration, we could exploit the parallelism even under a parallel-ready sequential call.[2] However, it seems that this pattern is used mostly when the code following the fork is to execute on some local data. If migration were used, the fork would be executed locally and the local code would be executed remotely. Instead, we represent the first fork as either an explicit seed, or if the arguments to the fork are possibly modified by the following code, as a full closure. We enqueue the seed (or closure) and start executing the following code. If no other processor stole the seed (or closure), then when the join is reached the parent suspends and the fork is executed.

The last pattern is the parallel loop. For this pattern an explicit seed is enqueued that points to a compiler generated code fragment which schedules iterations of the loop.

2.5 Discussion

In this section we have defined the requirements of the potentially parallel call. We introduce a control hierarchy of representations for nascent threads which frees our model from the necessity of migrating activation frames. We discuss how the return and scheduling mechanisms effect the cost of executing parallel-ready sequential calls and the implementation of the representations. Finally, we give examples of when the representations would be used. In the rest of the paper we explain how these concepts are actually implemented.

3 Storage Management: Stacklets

Analogous to the hierarchy of representations of the fork, we introduce a hierarchy of frame stores. Frames can be stored either on a stack, a stacklet, or a heap; depending upon the type of call and the requirements of the child. This hierarchy of frame stores allows the compiler to choose the least expensive storage mechanism for the activation frame.

The least versatile and least expensive method is to use a stack. This is only available for purely sequential calls, i.e., for calls that can never suspend. The most expensive method is to store each frame in a heap-allocated memory segment. Using a heap-allocated frame store requires links to be created from child to parent, which is implicit when using a stack.

Between these two extremes, we use stacklets for frames that are allocated for forks. A stacklet combines

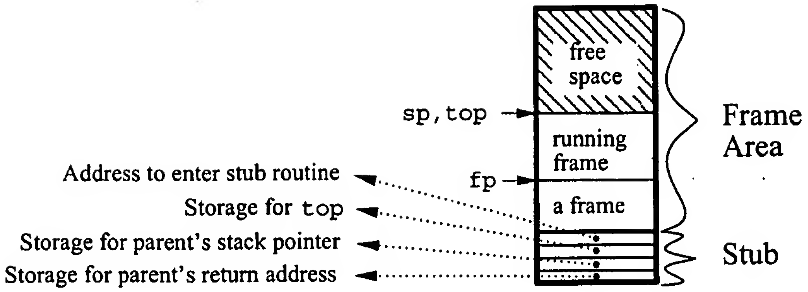
free space
sp, top
running frame
fp
a frame
Frame Area
Stub
Address to enter stub routine
Storage for top
Storage for parent's stack pointer
Storage for parent's return address

Figure 4: The basic form of a stacklet.

the efficiency of a stack and the flexibility of heap-allocated frames for those frames that are or may become independent threads of control. This method allows us to assign to each thread its own logically unbounded stack without having to manage multiple stacks or require garbage collection. Our compilation strategy allows us to use a sequential stack for sequential calls, stacklets for small to moderate sized forks, and the heap for large forks.

In the rest of this section, we describe the mechanics of stacklets and discuss their advantages over other intermediate methods of storing activation frames.

3.1 Stacklets

A *Stacklet* is a memory management primitive which efficiently supports cactus stacks. Each stacklet can be managed like a sequential stack. It is a region of contiguous memory on a single processor that can store several activation frames (see Figure 4). Each stacklet is divided into two regions, the stub and the frame area. The stub contains data that maintains the global cactus stack by linking the individual stacklets to each other. The frame area contains the activation frames.[3] In addition to a traditional stack pointer (`sp`) and frame pointer (`fp`), our model defines a *top pointer* (`top`) which—for reasons presented in the next section—points to the top of the currently used portion of the stacklet, or, in other words, to the next free location in the stacklet. These three pointers are kept in registers.

We recognize three kinds of calls—sequential call, fork, and remote fork—each of which maps onto a different kind of allocation request. A sequential allocation is one that requests space on the same stack as the caller. The child performs the allocation. Therefore it determines whether its frame can fit on the same stacklet. If so, `sp`, `fp`, and `top` are updated appropriately (see Figure 5). If not, a new stacklet is allocated and the child frame is allocated on the new stacklet (see Figure 6). This also happens for an eager fork,

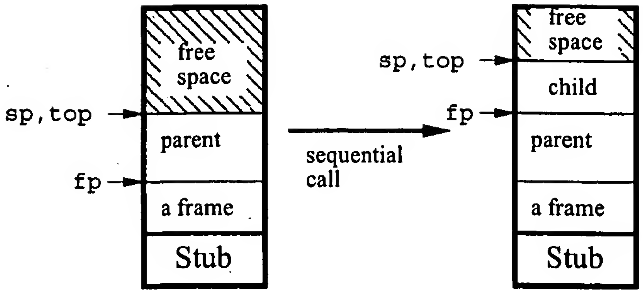

Figure 5: The result of a sequential call which does not overflow the stacklet.

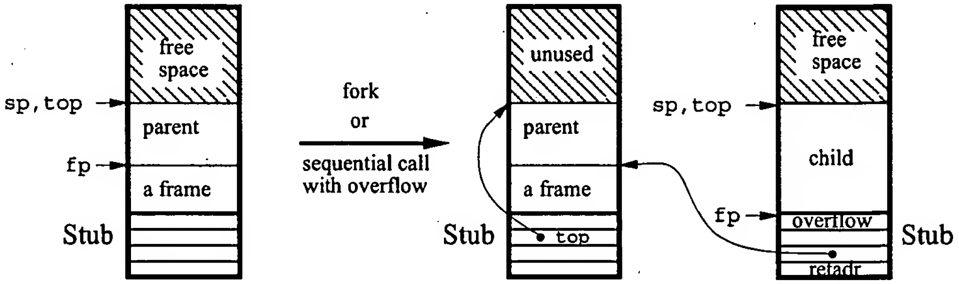

Figure 6: The result of a fork or of a sequential call which overflows the stacklet.

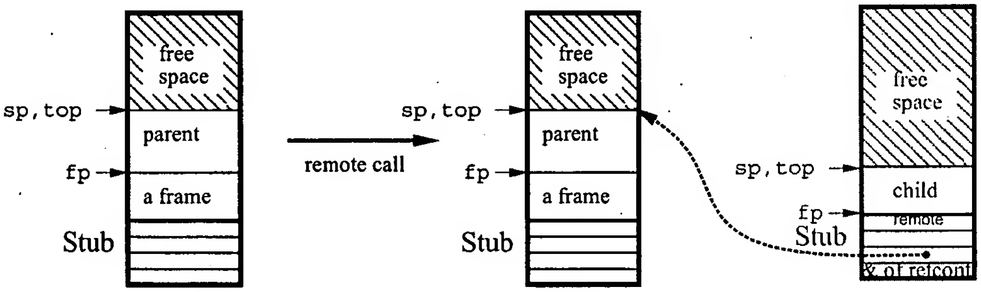

Figure 7: A remote fork leaves the current stacklet unchanged and allocates a new stacklet on another processor.

which causes a new stacklet to be created on the local processor. We can either run the child in the new stacklet immediately or schedule the child for later execution. In the former case, `fp`, `sp`, and `top` point to the child stacklet (see Figure 6). In the latter case, they remain unchanged after the allocation. For a remote fork there are no stacklet operations on the local processor. Instead, a message is sent to a remote processor with the child routine's address and arguments (see Figure 7).

The overhead in checking for stacklet overflow in a sequential call is two register-based instructions (an AND of the new `sp` and a compare to the old) and a branch (which will usually be successfully predicted). If the stacklet overflows, a new stacklet is allocated from the heap. This cost is amortized over the many invocations that will run in the stacklet.

3.2 Stacklet Stubs

Stub handlers allow us to use the sequential return mechanism even though we are operating on a cactus stack. The stacklet stub stores all the data needed for the bottom frame to return to its parent. When a new stacklet is allocated, the parent's return address and frame pointer are saved in the stub and a return address to the stub handler is given to the child. When the bottom frame in a stacklet executes a return, it does not return to its caller. Instead it returns to the stub handler. The stub handler performs stacklet deallocation and, using the data in the stacklet stub, carries out the necessary actions to return control to the parent (restoring `top`, and making `sp` and `fp` point to the parent).

In the case of a remote fork, the stub handler uses indirect active messages [38] to return data and control to the parent's message handler, which in turn is responsible for integrating the data into the parent frame and indicating to the parent that its child has returned.

3.3 Compilation

To reduce the cost of frame allocation even further, we construct a call graph which enables us to determine for all but the recursive calls whether an overflow check is needed [15]. Each function has two entry points, one that checks stacklet overflow and another that does not. If the compiler can determine that no check is needed, it uses the latter entry point. This analysis inserts preventive stacklet allocation to guarantee that future children will not need to perform any overflow checks.

3.4 Discussion

Stacklets provide efficient storage management for parallel execution. In the next section we will see that potentially parallel calls can use the same efficient mechanism as regular sequential calls, because each stacklet preserves the invariants of a stack. Specifically, the same call and return mechanisms are used, so arguments and results can be passed in registers. These benefits are obtained at a small increase to the cost of sequential calls made in the stacklet, namely checking whether a new stacklet needs to be allocated in the case of an overflow or parallel call. The extra cost amounts to a test and branch along with the use of an additional register. This overhead is required only when the compiler cannot statically determine that no check is needed. Stubs eliminate the need to check for underflows. This contrasts with previous approaches which always require some memory references or a garbage collector.

We can further reduce the cost of using stacklets by utilizing all three levels of our frame store hierarchy. Purely sequential calls, i.e., a call to a child which is guaranteed not to suspend in itself or in any of its descendants, can be executed on a sequential stack which has no overhead. Parallel-ready sequential calls and sequential calls that cannot be proven not to suspend are made on stacklets. Using this paradigm, we use one register (a sequential stack pointer) for the sequential stack and two for the cactus stack (a stacklet stack pointer and a stacklet top pointer).

The main disadvantage in using stacklets is that if a frame is at the top of a stacklet, each of its children will have to be allocated on a new stacklet. This boundary case is similar to the overflow case when using register windows. We can reduce the probability that this happens by increasing the size of each stacklet so that it happens rarely. Also, the compiler optimization mentioned above can eliminate many of the places where this could happen.

4 A Fast Thread Fork

In this section we show how to maintain the efficiency and invariants of a sequential invocation with the flexibility of a parallel one. First, we describe the call mechanism. Then we show how to combine two of the three independent requirements for return. Finally, we describe a method that allows us to combine all three.

4.1 The Parallel-Ready Sequential Call

Our goal is to make a fork as fast as a sequential call when the forked child executes sequentially. Using stacklets as the underlying frame-storage allocation mechanism gives us a choice of where to run the new thread invoked by the fork. The obvious approach is to explicitly fork the new thread using the parallel allocation explained in the previous section. However, if the child is expected to complete without suspending, i.e., if it behaves like a sequential call, we would rather treat it like a sequential call and invoke the child on the current stacklet.

This section defines the parallel-ready sequential call (`prscall`) which behaves like a sequential call unless it suspends, in which case, in order to support the logical parallelism implied by the fork it represents, it directly resumes the parent and behaves like an eagerly forked thread. `prscall` behaves like a sequential

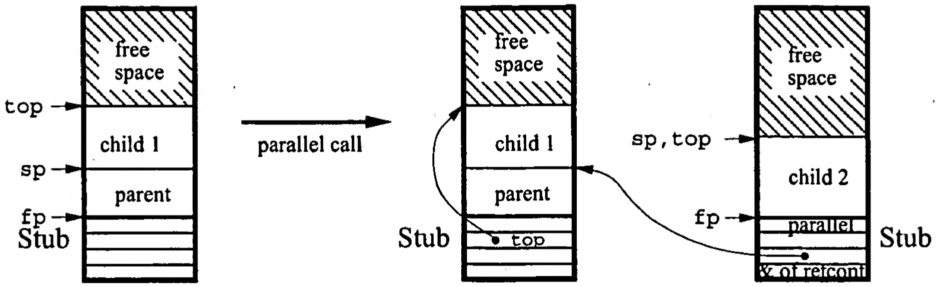

Figure 8: A parallel call creates a new stacklet.

call in that it transfers control (and its arguments) directly to the new thread. Furthermore, if the new thread completes without suspending, it returns control (and results) directly to its parent.

If the child suspends, it must resume its parent in order to notify its parent that the `prscall` really required its own thread of control. Thus, the child must be able to return to its parent at either of two different addresses, one for normal return and one for suspension. Instead of passing the child two return addresses, the parent calls the child with a single address from which it can derive both addresses. At the implementation level, this use of multiple return addresses can be thought of as an extended version of continuation passing [2], where the child is passed two different continuations, one for normal return and one for suspension. The compiler ensures that the suspension entry point precedes the normal return entry point by a fixed number of instructions. In the case of suspension, the compiler uses simple address arithmetic to calculate the suspension entry point.

If the child suspends, the parent will not be the topmost frame in the stacklet, i.e., `sp` will not equal `top` (the situation shown in Figure 8). To maintain the sequential stack invariant, we do not allocate future children of the parent on the current stacklet. Instead, while there is a suspended child above the parent in the current stacklet, we allocate future children, sequential or parallel, on their own stacklets (see Figure 8). As a result, the translation for a fork which immediately follows another fork must first compare `sp` and `top`. If they are equal, the call occurs on the current stacklet (as in Figure 5). If they are different, it starts a new stacklet (as in Figure 8). As a result, regardless of the children's return order, no stacklet will ever contain free space between allocated frames, thus simplifying memory management.

In summary, `prscall` allows a potentially parallel thread to be executed sequentially and still have the ability to suspend and obtain its own thread of control. A child that needs its own thread of control takes over its parent thread, causing its parent to allocate subsequent work on other stacklets. Using stacklets and compiler support we have created a multithreaded environment in a single address space which gives each thread a logically unbounded stack.

4.2 Parent Controlled Return Continuations

In this section we show how we reduce the overheads that arise from a child returning to its parent. As described in Section 2.1, there are three independent actions that must be carried out on return: the inlet must be executed, the synchronization state updated, and finally control must be transferred to the continuation address. Here we show how we can eliminate the need to update the synchronization state on every return and how we can combine the continuation address with the inlet.

We begin by observing that if a child suspends, we can no longer use the originally supplied return address when the child is resumed. For, once the child has suspended, the child and parent are truly separate threads, and the parent may have already carried out the work immediately following the fork that created the child. Thus, we need some way to modify the child's return address so that the child will return to a location of the parent's choosing. Since we are using stacklets, both the parent and the child know where the child's return address is located in the stack. If the parent is given permission to change the return address, it can change it to reflect the new state of the parent's computation. The new return address will return the child to the point in the parent function that reflects that the child, and any work initiated after the child suspended, have been carried out. We call this mechanism *parent controlled return continuations* (PCRCs).

As described in the previous section, every child is given two addresses, one for normal return and one for suspension. If the child returns without ever having suspended the normal return address points to an inlet which immediately follows the fork of the child in the parent, just like a sequential call. As long as none of the children suspend we do not update any synchronization counters, nor do we store a continuation address.

When a child suspends, the code at the suspension address performs three tasks. First, it updates the synchronization counter, as it would have before an eager parallel call. Next, using PCRCs, it changes the child's return address to point to a suspended-inlet (described below). Finally, it transfers control to the statement following the original fork.

For every `prscall` the compiler creates two sets of inlets: one set for normal return and the suspended-inlet set, used for a child that has at one time suspended. The code at the suspend address for a suspended-inlet does nothing but suspend the parent. The code at the return address for the suspended-inlet stores the results, updates the synchronization counter to indicate that the child has completed, and then jumps directly to the join following the forks.

The join following the forks checks if any children are outstanding. If they are, it suspends the parent, otherwise it continues execution in the parent. Thus, before the first fork is executed, we must reset the synchronization counter to indicate no calls are outstanding. Instead of always touching the synchronization counter twice for each fork (once before execution and once on completion), we only have to reset it before any of the forks have executed and update it for forks that suspend.

By using PCRCs and by duplicating the inlet code (which is normally a single store), we are able to eliminate almost all the overhead usually associated with parallel calls. When a child does suspend, the extra cost of transforming it into its own independent thread is no more than if we had executed it as an independent thread in the first place.

4.3 Efficient Synchronization: Synchronizers

We minimize the synchronization cost due to joins by extending the use of PCRCs with judicious code duplication and by exploiting the flexibility of the indirect jump in the return instruction. The basic idea is to change the return continuation of a child to reflect the synchronization state of the parent. In this way, neither extra synchronization variables nor tests are needed. The amount of code duplicated is small since we need to copy only the inlet code fragments. This allows us to combine the return with synchronization at no additional run-time cost. Here we describe the synchronizer transformations for two forks followed by a join.[4]

In the previous section we showed how we were able to encode the inlet address and continuation address in the return continuation. The return continuation for a child was changed at most once by the parent if the child suspended. Here we extend the encoding to include the synchronization state of the parent. If none of the children suspend, then each will return and the parent will call the next child until the last child returns and the parent executes the code following the join.

If any child suspends, then not only does the inlet for that child have to change to reflect a new continuation address, but each following fork must be supplied a return continuation that indicates that there is an outstanding child. Furthermore, when a child that previously suspended returns, it must update

```
Fork X
Fork Y
Join
continue after join
```

Main program text

```
    pcrc(X)←inlet1A
    Call X
inlet1A.susp:
    pcrc(X)←inlet1C
    pcrc(Y)←inlet2B
    Fork Y
inlet1A.ret:

    pcrc(Y)←inlet2A
    Call Y
inlet2A.susp:
    suspend();
inlet2A.ret:
done:
    continue after join
```

Inlets which handle return and suspension.

```
inlet1B.susp:
    suspend();
inlet1B.ret:
    goto done;

inlet1C.susp:
    suspend();
inlet1C.ret:
    pcrc(Y)←inlet2A
    suspend();

inlet2B.susp:
    suspend();
inlet2B.ret:
    pcrc(X)←inlet1B
    suspend();
```

Auxiliary inlets which handle synchronization.

Figure 9: An example of applying the synchronizer transformation to two forks followed by a join. Each inlet, *i*, is represented by two labels, a suspension label (*i.susp*) and a return label (*i.ret*). In the pseudocode, we expose the operations that set a child's return address; *pcrc(X)* denotes the return address for the child *X*.

the return continuations for each of the outstanding children to reflect the new synchronization state. In the worse case this can require O(n^2) stores for n forks. For this reason we only apply the synchronizer transformation when $n = 2$.

In the case where two forks are followed by a join, the transformation is straightforward and cost effective. As shown in Figure 9, the synchronization transformation generates five inlets for two forks: the first two (in the middle of the figure) are used when no suspension occurs and the last three (to the right of the figure) are used when suspension does occur.

There are three cases to consider for the first fork. First, if it returns without suspending, it returns to an inlet (inlet 1A) that starts the second fork. Second and third, if it has suspended, then when it returns either the second call will have been completed, or it will still be executing. In the former case its inlet (inlet 1B) should continue with the code *after* the join. In the latter, it should suspend the parent (inlet 1C).

There are two cases to consider for the second fork. Either the first call is completed when it returns, or the first call is outstanding. If the first call is completed, then the inlet (inlet 2A) continues with the code after the join. If the first call is still outstanding, then the inlet (inlet 2B) changes the PCRC for the first call to inlet 1B and suspends the parent.

When no suspensions occur, inlets 1A and 2A are executed, and the runtime cost of synchronization is zero. If the first fork suspends, it starts the second fork with inlet 2B and sets its own inlet to 1C. If call 1 (2) returns next, it will set the inlet for call 2 (1) to inlet 2A (1B). This allows the last call that returns to immediately execute the code after the join without performing any tests. In this case, the runtime synchronization cost is two stores, which is less than the cost of explicit synchronization.

5 Representing Potentially Parallel Work

So far we have shown how to reduce the overhead of potential parallelism when it actually unfolds sequentially on the local processor. Here we extend these ideas to show how to represent potentially parallel work for remote processors. In doing so, we describe the remaining representations for the potentially parallel call: thread seeds and the full closure. These representations allow us reduce the overhead of the potentially parallel call without restricting the logical parallelism available in the program. In other words, they allow us to represent the potential work in the system so that it may be distributed efficiently among multiple processors. Our goal is to allow work to be distributed remotely but to pay the cost only when there is an actual need to do so.

5.1 Thread Seeds

Thread seeds are a direct extension of the multiple return addresses we introduced in Section 4.1 to handle the suspension of children invoked by a parallel-ready sequential call. As shown in Figure 10, a thread seed is a code fragment with three entry points: one for child return, one for child suspension, and one for an external work-stealing request. When the compiler determines that there is work that could be run in parallel, it creates a thread seed which represents the work. For example, with two successive forks, the fork for the first thread will be associated with a thread seed representing the fork of the second thread.

We combine seed generation and fork into a single primitive, `pcall X,` S_Y, where `X` is the function to call and S_Y is a thread seed that will, when executed in the context of the parent, cause the function `Y` to be invoked. Upon execution of the `pcall`, a seed is created and control is transferred to `X`, making the current (parent) frame inactive. The newly created seed remains dormant until one of three things happens: the child returns (the seed is assumed), the child suspends (the seed is activated), or a remote processor requests work (the seed is stolen). All three cases require the intervention by the parent. If the child returns, the parent picks up the seed and inlines the new thread of control into its own, i.e. `Y` executes on the parent's stacklet, just as if it had been called sequentially. If the child suspends, the parent activates the seed by

```
                         pushSeed(S_y);
PCALL X, S_y             call X();
                         return address is by convention assumption

                     stealing:
                         change X's return continuation
The Seed: S_y            goto invokeYremotely;
                     suspension:
                         change X's return continuation
All seeds are a fixed    goto invokeY;
number of            assumption:
instructions.            popSeed();
```

Figure 10: How `pcall` and thread seeds are implemented.

spawning `Y` off on its own new stacklet. The seed becomes a new thread concurrent with the first thread, but on the same processor. If a remote processor requests work, the parent executes a remote call of `Y`, which becomes a new thread running concurrently with the current thread but on another processor.

When a remote work request is received, the run-time system must somehow find the thread seed (the third continuation) to initiate the creation of remote work. Here we consider two approaches to finding such work: an implicit seed model and an explicit seed model.

In the implicit seed model, the remote work request interrupts the child, which then continues execution at the work-stealing entry point of its parent. If there is no work, the entry point contains a code fragment to jump to the parents ancestor on the stack. The search continues until either work is generated or no work is found because no excess parallelism is present. For the implicit model, the `pushSeed` and `popSeed` macros in Figure 10 turn into no-ops and the planting of a seed is an abstract operation. The advantage of this model is that when a `pcall` is made no bookkeeping is required. Instead, the stack frames themselves form the data structure needed to find work. The disadvantage is that finding work is more complex.

In the explicit seed model, when a seed is planted a special continuation is pushed onto the top of a *seed queue*. The continuation is a pointer to the return address in the frame. The calling convention is such that the return from the child defaults to the assumption point. If a child suspends, it saves the top pointer in the stacklet stub, pops the top seed off the queue, sets the `sp` as indicated by the seed, and jumps into the suspension entry point of the seed.

The explicit queueing of seeds allows us to find work with just a few instructions. For instance, if a child suspends, it can find its ancestor, which has more work to perform, merely by popping off the top seed. Or, more importantly, if a remote processor requests work, we can determine if there is work by simply comparing the top and bottom pointers to the seed queue. We can also spawn off that work by jumping through the work-stealing entry point of the seed at the bottom of the queue. The parent, invoked through the seed, will execute the work-stealing routine, placing any appropriate seed on the bottom of the queue. The drawback of this scheme is that even when a seed is inlined into the current thread (the sequential case) there is an extra cost of two memory references over the previously described implicit scheme.

5.2 Costs

We define the overhead cost of a representation as the extra cost it introduces above that of a sequential call. Since any of these calls may end up executing in parallel we define a secondary cost, the transformation cost, as the cost of converting the call to run in parallel versus executing it in parallel *ab initio*. As we shall see, most of these representations have very small transformation costs, while their basic cost varies from zero to the cost of an explicit parallel call (minus the base cost of a sequential call).

There are three aspects that make up the cost of a representation: creation, linkage, and instantiation. Creation cost refers to the cost of creating the nascent form of the call which can later be instantiated into an executing call. Linkage refers to the cost of enqueueing the representation so that it can later be found and executed. The instantiation cost represents the cost of turning the representation into an executing call. Clearly the eager explicit parallel call and the sequential call have no creation or linkage costs. For these two representations the instantiation phase is the cost of either creating a new thread or a new sequential child. These costs depend on the underlying runtime structures used to represent threads, stacks, etc.

The transformation cost has two components: direct and indirect. The direct part is the cost of converting a sequential call into a parallel call. The indirect component is the cost that the conversion adds to the parent of the call being converted (and the parent's future children).

5.2.1 The Parallel-Ready Sequential Call

The parallel-ready sequential call has minimal overhead cost. It is the cheapest method that can be used to implement a fork without additional knowledge.[5] Like the sequential call, this implementation has no creation or linkage cost. As described in Section 4.3, the best case execution overhead is also zero. In the worst case, synchronization overhead will add $3/n$ memory references, a compare, and branch where n is the number of forks in the forkset.

The direct transformation cost is zero; it costs the same to suspend a parallel thread as a parallel-ready sequential call. The indirect cost depends on the frame storage method. If the child was implemented in a heap based frame, it is zero. Since the suspended child steals the parent's frame, if it is allocated on a stacklet, it will cause all future children in the forkset to be allocated on a separate stacklet, even though they may run sequentially.

5.2.2 The Implicit Seed

Both the creation and linkage costs of the implicit seed are zero, as it is created implicitly by the return address of a parallel-ready sequential call. Implicit seeds work very well in handling the problem of suspension. When a sequentially called fork suspends, it uses the return address it has to locate and execute the implicit seed. However, if a remote processor wants to steal work it will have to (in the general case) do a tree walk of all outstanding frames to find an implicit seed. Its direct transformation cost is zero when a parallel-ready sequentially called child suspends and O(number of frames) when a remote processor wants work.

5.2.3 The Explicit Seed

Like the implicit seed the creation cost of the explicit seed is zero, as the seed is implemented by a compiler generated code fragment. However, instead of using the preceding call's return address for linkage, it is pushed onto a queue. Thus it costs nothing to create, but it incurs a small linkage cost, that of enqueueing it and later removing it if the fork it represents ends up executing sequentially. However, the direct transformation cost is that of a dequeue independent of the reason.

5.2.4 The Full Closure

The closure's creation cost is proportional to the number of arguments to the fork. Like the explicit seed it must be enqueued and has a linkage cost of one enqueue and one dequeue. The transformation cost is also non-zero since if it were initially executed in parallel, we would not have needed to create the closure. However, the overhead cost is low compared to an eagerly executed parallel call that could have been executed sequentially.

6 Experimental Results

In this section we present preliminary performance results for our techniques on both uni- and multiprocessors. Our uniprocessor data were collected on a SparcStation 10. Our multiprocessor data were collected on a CM-5.

We have produced a parallel version of C for the CM-5 which incorporates the techniques presented in this paper. To evaluate these techniques we begin by comparing the performance of four different implementations of the doubly recursive Fibonacci function; which does essentially nothing but function invocation. As shown in Table 1, the C version is significantly slower than either the synchronizer or the seed version. The reason is that our stacklet management code does not use register windows, which introduce a high overhead on the Sparc. For a fair comparison we wrote an assembly version of Fib that also does not use register windows. This highly optimized assembly version runs only 18% faster than the synchronizer version, which incorporates all the mechanisms for multithreading support.

Next, we look at the efficiency of work-stealing combined with seeds on a parallel machine by examining the performance of the synthetic benchmark proposed in [26] and also used in [39]. Grain is a doubly recursive program that computes a sum, where each leaf executes a loop of g instructions, thus allowing us to control the granularity of the leaf nodes. We compare its efficiency to that of sequential C code compiled by gcc. As shown in Figure 11, using stacklets we achieve over 90% efficiency when the grain size is as little as 400 cycles. Compare this to the grain size of an invocation of fib, which is approximately 30 cycles. Most of the inefficiency comes from the need to poll the CM-5 network. The speed-up curve in Figure 11 shows that even for very fine-grained programs, the thread seeds successfully exploit the entire machine.

Further evidence that lazy threads are efficient is presented in Table 2, where we compare our lazy thread model with TAM for some larger programs on the Sparc. At this time our Id90 compiler uses a primitive version of explicit seed creation. In addition to the primitives described so far, the compiler uses strands, a mechanism to support fine-grained parallelism within a thread [14]. We see a performance improvement ranging from 1.1 times faster for coarse-grained programs, like blocked matrix multiply (MMT), to 2.7 times faster for more finely-grained programs. We expect an additional benefit of up to 30% when the compiler generates code using synchronizers.

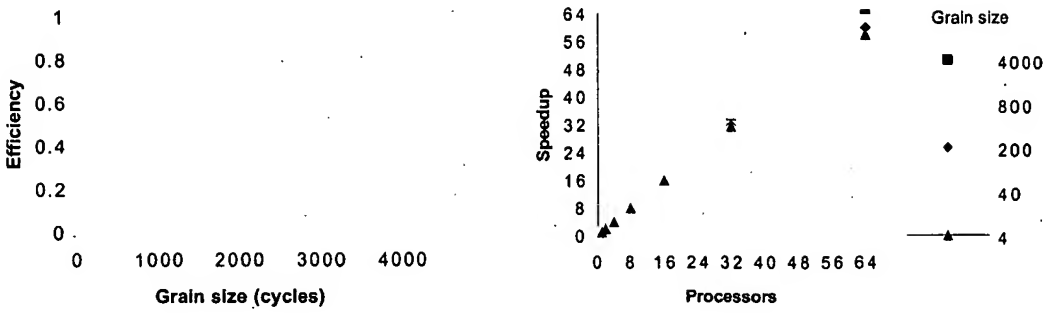

Figure 11: Efficiency of lazy threads on the CM-5 compared to the sequential C implementation as a function of granularity. We use the synthetic benchmark Grain [26, 39].

7 Related Work

Attempts to accommodate logical parallelism include thread packages [11, 34, 7, 16], compiler techniques and clever run-time representations [9, 30, 26, 39, 37, 33, 17], and direct hardware support for fine-grained parallel execution [19, 3]. These approaches have been used to implement many parallel languages, e.g. Mul-T [23], Id90 [9, 30], CC++ [6], Charm [20], Opus [25], Cilk [4], Olden [5], and Cid [29]. The common goal is to reduce the overhead associated with managing the logical parallelism. While much of this work overlaps ours, none has combined the techniques described in this paper into an integrated whole. More importantly, none has started from the premise that all calls, parallel or sequential, can be initiated in the exact same manner.

Our work grew out of previous efforts to implement the non-strict functional language Id90 for commodity parallel machines. Our earlier work developed a *Threaded Abstract Machine* (TAM) which serves as an intermediate compilation target [9]. The two key differences between this work and TAM are that under TAM calls are always parallel, and due to TAM's scheduling hierarchy, calling another function does not immediately transfer control.

Our lazy thread fork allows all calls to begin in the same way and creates only the required amount of concurrency. Many other researchers have proposed schemes which deal lazily with excess parallelism. One of the simplest schemes is *load based inlining*, which uses load characteristics of the parallel machine to decide at the time a potentially parallel call is encountered whether to execute it sequentially (inline it) or execute it in parallel [23]. This has the advantage of dynamically increasing the granularity of the program. However, these decisions are irrevocable, which can lead to serious load imbalances or deadlock. Our approach builds on *lazy task creation* (LTC) which maintains a data structure to record previously encountered parallel calls [26]. When a processor runs out of work, dynamic load balancing can be effected by stealing previously created lazy tasks from other processors. These ideas were studied for Mul-T running on shared-memory machines. The primary difference is that LTC always performs extra work for parallel calls, whether they execute locally or remotely. Even the lazy tasks that are never raised to full fledged tasks are "spawned off" in the sense that they require extra bookkeeping. Since work-stealing causes the parent to migrate to a new thread, LTC depends on the ability of the system to migrate activation frames. This either requires shared-memory hardware capabilities or restricts the the kind of pointers allowed in the language. Our implementation works on both distributed- and shared-memory systems. Finally, LTC also depends on a garbage collector, which hides many of the costs of stack management.

Another proposed technique for improving LTC is *leapfrogging* [39]. Unlike the techniques we use, it restricts the behavior of the program in an attempt to reduce the cost of futures. Leapfrogging has been implemented using Cthreads, a light-weight thread package. Another interesting approach, based on SML/NJ, represents threads by simple continuations [27], which in turn can be represented efficiently. Since continuations are supported by the language model, important aspects of the thread system, such as scheduling and synchronization, can be described in the language itself. This system can be used to implement user-level thread packages directly within the ML language

We use stacklets for efficient stack-based frame allocation in parallel programs. Previous work in [17] describes similar ideas for handling continuations efficiently. Olden [5] uses a "spaghetti stack." In both systems, the allocation of a new stack frame always requires memory references and a garbage collector. Olden's thread model is more powerful than ours, since in Olden threads can migrate. The idea is that a thread computation which is following references to unstructured heap-allocated data might increase locality if migration occurs [32]. On the other hand, this model requires migrating the current call frame as well as disallowing local pointers to other frames on the stack.

StackThreads [35] uses both a stack and the heap for storing activation frames in an attempt to reduce overhead for fine-grained programs running on a single processor. Activation frames are initially placed on the stack and if they block, they are moved onto the heap. Since the sequential call invariants are not enforced StackThreads does not take advantage of passing control and data at the same time, reducing register usage and increasing synchronization overhead. They take a diametrically opposing point of view in that all calls, sequential or parallel, use the same representation. This triples the direct function call/return overhead and prevents the use of registers.

A much simpler thread model is advocated in Shared Filaments [10] and Distributed Filaments [13]. A filament is a very lightweight thread which does not have a stack associated with it. This works well when a thread does not fork other threads. More general threads are supported with a single stack because language restrictions make it impossible for a parent to be scheduled when any of its children are waiting for a synchronization event. Distributed filaments are combined with a distributed shared memory system. In the case of a remote page fault, communication can be overlapped with computation because each processor runs multiple scheduling threads.

The way thread seeds encode future work builds on the use of multiple offsets from a single return address to handle special cases. This technique was used in SOAR [36]. It was also applied to Self, which uses parent controlled return continuations to handle debugging [18]. We extend these two ideas to form synchronizers.

Finally, many light-weight thread packages have been developed. Cthreads is a run-time library which provides multiple threads of control and synchronization primitives for parallel programming at the level of the C language [7]. Scheduler activations reduce the overhead by moving fine-grained threads completely to the user level and relying on the kernel only for infrequent cases [1]. Synthesis is an operating systems kernel for a parallel and distributed computational environment which is interesting in our context because it integrates dynamic load balancing capabilities and applies dynamic compilation techniques [24]. Chant [16] is a lightweight threads package which is used in the implementation of an HPF extension called Opus [25]. Chant provides an interface for lightweight, user-level threads which have the capability of communication and synchronization across separate address spaces. While user-level thread packages eliminate much of the overhead encountered in traditional operating systems thread packages, they are still not as light-weight as many of the systems mentioned above which use special run-time representations supported by the compiler. Since the primitives of thread packages are exposed at the library level, compiler optimizations presented in this paper are not possible for such systems.

8 Summary

We have shown that by integrating a set of innovative techniques for call frame management, call/return linkage, and thread generation we can provide a fast parallel call which obtains nearly the full efficiency of a sequential call when the child thread executes locally and runs to completion without suspension. This occurs frequently with aggressively parallel languages such as Id90, as well as more conservative languages such as C with parallel calls.

The central idea is to pay for what is used. Thus, a local fork is performed essentially as a sequential call, with the attendant efficient stack management and direct transfer of control and data. The only preparation for parallelism is the use of bounded-size stacklets and the provision of multiple return entry points in the parent. If the child actually suspends before completion, control is returned to the parent so that it can take appropriate action. Similarly, remote work is generated lazily. When a thread has work that can be performed remotely, it exposes an entry point, called a thread seed, that will produce the remote work on demand. If the work ends up being performed locally, it is simply inlined into the local thread of control as a sequential call. We exploit the one bit of flexibility in the sequential call, the indirect jump on return, to provide very fast synchronization and to avoid explicit checking for special cases, such as stacklet underflow.

Empirical studies with a parallel extension to C show that these techniques offer very good parallel performance and support fine-grained parallelism even on a distributed memory machine. Integrating these methods into a prototype compiler for Id90, depending on the frequency of parallel calls in the program, results in an improvement of nearly a factor of two over previous approaches.

Acknowledgements

Computational support at Berkeley was provided by the NSF Infrastructure Grant number CDA-8722788. Seth Copen Goldstein is supported by an AT&T Graduate Fellowship. Klaus Erik Schauser received research support from the Department of Computer Science at UCSB. David Culler is supported by an NSF Presidential Faculty Fellowship CCR-9253705 and LLNL Grant UCB-ERL-92/172.

Tables

Table 1: Comparing runtimes of fib 31 on a SparcStation 10.

	Runtime
Compilation Method	(secs)
gcc -O4 fib.c	2.29
Assembly version of Fib	1.22
Fib with stacklets, lazy threads, and synchronizers	1.50
Fib as above with explicit seeds	1.86

Table 2: Dynamic run-time in seconds on a SparcStation 10 for the Id90 benchmark programs under the TAM model and lazy threads with multiple strands using explicit seeds. The programs are described in [9].

Program	Short Description	Input Size	TAM	Lazy Threads
Gamteb	Monte Carlo neutron transport	40,000	220.8	139.0
Paraffins	Enumerate isomers of paraffins	19	6.6	2.4
Simple	Hydrodynamics and heat conduction	1 1 100	5.0	3.3
MMT	Matrix multiply test	500	70.5	66.5

Footnotes

[1] This is similar to what is provided in many kernel threads packages. Our threads, however, are stronger than those in TAM [9] and in some user-level threads packages, e.g. Chorus [34], which require that the maximum stack size be specified upon thread creation so that memory can be preallocated.

[2] Another approach would have the compiler package up the following code as a separate function and use the 2-way fork pattern. But this suffers from the same problems as migration.

[3] Frames that are larger than a single stacklet are stored on the heap.

[4] Although the transformation is general enough to be applied to n forks, it does not seem cost effective.

[5] It should be noted that in some rarely occurring cases an eager fork can be cheaper than a parallel-ready sequential call.

References

[1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1), February 1992.

[2] A. W. Appel. *Compiling with continuations*. Cambridge University Press, New York, 1992.

[3] Arvind and D. E. Culler. Dataflow architectures. In *Annual Reviews in Computer Science*, volume 1, pages 225–253. Annual Reviews Inc., Palo Alto, CA, 1986.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, P. Lisiecki, K. H. Randall, A. Shaw, and Y. Zhou. *Cilk 1.1 reference manual*. MIT Lab for Comp. Sci., 545 Technology Square, Cambridge, MA 02139, September 1994.

[5] M.C. Carlisle, A. Rogers, J.H. Reppy, and L.J. Hendren. Early experiences with Olden (parallel programming). In *Languages and Compilers for Parallel Computing. 6th International Workshop Proceedings*, pages 1–20. Springer-Verlag, 1994.

[6] K.M. Chandy and C. Kesselman. Compositional C++: compositional parallel programming. In *Languages and Compilers for Parallel Computing. 5th International Workshop Proceedings*, pages 124–44. Springer-Verlag, 1993.

[7] E. C. Cooper and R. P. Draves. C-Threads. Technical Report CMU-CS-88-154, Carnegie-Mellon University, February 1988.

[8] D. Culler, A. Dusseau, S. Goldstein, S. Lumetta, T. von eicken, and K. Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing 93*, pages 262–273, November 1993.

[9] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken. TAM — a compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, 18:347–370, July 1993.

[10] D. R. Engler, D. K. Lowenthal, and Andrews G. R. Shared Filaments: efficient fine-grain parallelism on shared-memory multiprocessors. Technical Report TR 93-13a, University of Arizona, April 1993.

[11] J.E. Faust and H.M. Levy. The performance of an object-oriented threads package. In *SIGPLAN Notices*, pages 278–88, Oct. 1990.

[12] Message Passing Interface Forum. Mpi: a message-passing interface standard. *International Journal of Supercomputer Applications and High Performance Computing*, vol.8,(no.3-4):169–416, Fall-Winter 1994.

[13] V.W. Freeh, D.K. Lowenthal, and G.R. Andrews. Distributed Filaments: efficient fine-grain parallelism on a cluster of workstations. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 201–13. USENIX Assoc, 1994.

[14] S. C. Goldstein, D. E. Culler, and K. E. Schauser. Lazy Threads, Stacklets, and Synchronizers: Enabling primitives for compiling parallel languages. Technical report, University of California at Berkeley, 1995.

[15] D. Grunwald, B. Calder, S. Vajracharya, and H. Srinivasan. Heaps o' Stacks: combined heap-based activation allocation for parallel programs. URL: http://www.cs.colorado.edu/~ grunwald/, 1994.

[16] M. Haines, D. Cronk, and P. Mehrotra. On the design of Chant: a talking threads package. In *Proceedings Supercomputing '94 (Cat. No.94CH34819)*, pages 350–9. IEEE Comput. Soc. Press, 1994.

[17] R. Hieb, R. Kent Dybvig, and C. Bruggeman. Representing control in the presence of first-class continuations. In *SIGPLAN Notices*, pages 66–77, June 1990.

[18] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *SIGPLAN Notices*, pages 32–43, July 1992.

[19] H. F. Jordan. Performance measurement on HEP — a pipelined MIMD computer. In *Proc. of the 10th Annual Int. Symp. on Comp. Arch.*, Stockholm, Sweden, June 1983.

[20] L.V. Kale and S. Krishnan. CHARM++: a portable concurrent object oriented system based on C++. In *SIGPLAN Notices*, pages 91–108, Oct. 1993.

[21] V. Karamcheti and A. Chien. Concert: efficient runtime support for concurrent object-oriented programming languages on stock hardware. In *Proceedings SUPERCOMPUTING '93*, pages 598–607. IEEE Comput. Soc. Press, Nov. 1993.

[22] C. koelbel, D. Loveman, R. Schreiber, G. Steel Jr., and M. Zosel. *The High Performance Fortran Handbook.* The MIT Press, 1994.

[23] D.A. Kranz, Jr. Halstead, R.H., and E. Mohr. Mul-T: a high-performance parallel Lisp. In *SIGPLAN Notices*, pages 81–90, July 1989.

[24] H. Massalin and C. Pu. Threads and input/output in the Synthesis kernel. In *Twelfth ACM Symposium on Operating Systems Principles*, December 1989.

[25] P. Mehrotra and M. Haines. An overview of the Opus language and runtime system. In *Languages and Compilers for Parallel Computing. 7th International Workshop Proceedings*, pages 346–60. Springer-Verlag, 1995. AN4917658.

[26] E. Mohr, D.A. Kranz, and Jr. Halstead, R.H. Lazy task creation: a technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, vol.2,(no.3):264–80, July 1991.

[27] J. G. Morrisett and A. Tolmach. Procs and Locks: a portable multiprocessing platform for standard ML of New Jersey. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[28] R. S. Nikhil. Id (version 88.0) reference manual. Technical Report CSG Memo 284, MIT Lab for Comp. Sci., March 1988.

[29] R. S. Nikhil. Cid: A parallel, "shared memory" C for distributed-memory machines. In *Languages and Compilers for Parallel Computing. 7th International Workshop Proceedings*. Springer-Verlag, 1995.

[30] R.S. Nikhil. A multithreaded implementation of Id using P-RISC graphs. In *Languages and Compilers for Parallel Computing. 6th International Workshop Proceedings*, pages 390–405. Springer-Verlag, 1994.

[31] M.D. Noakes, D.A. Wallach, and W.J. Dally. The J-Machine multicomputer: an architectural evaluation. In *Computer Architecture News*, pages 224–35, May 1993.

[32] A. Rogers, M.C. Carlisle, J.H. Reppy, and L.J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Transactions on Programming Languages and Systems*, vol.17,(no.2):233–63, March 1995.

[33] A. Rogers, J. Reppy, and L. Hendren. Supporting SPMD execution for dynamic data structures. In *Languages and Compilers for Parallel Computing. 5th International Workshop Proceedings*, pages 192–207. Springer-Verlag, 1993.

[34] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Overview of the CHORUS distributed operating system. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–69. USENIX Assoc, 1992.

[35] K. Taura, S. Matsuoka, and A. Yonezawa. StackThreads: an abstract machine for scheduling fine-grain threads on stock CPUs. In *Theory and Practice of Parallel Programming. International Workshop TPPP '94. Proceedings*, pages 121–36. Springer-Verlag, 1995. AN4986592.

[36] D. M. Ungar. *The design and evaluation of a high performance Smalltalk system.* ACM distinguished dissertations. MIT Press, 1987.

[37] M.T. Vandevoorde and E.S. Roberts. WorkCrews: an abstraction for controlling parallelism. *International Journal of Parallel Programming*, vol.17,(no.4):347–66, Aug. 1988.

[38] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a mechanism for integrated communication and computation. In *Proc. of the 19th Int'l Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[39] D.B. Wagner and B.G. Calder. Leapfrogging: a portable technique for implementing efficient futures. In *SIGPLAN Notices*, pages 208–17, July 1993.

[40] A. Yonezawa. *ABCL– an object-oriented concurrent system* . MIT Press series in computer systems. MIT Press, 1990.

List of Figures